Polynomial root finding using iterated eigenvalue computation

Steven Fortune
Bell Laboratories
sjf@research.bell-labs.com

Abstract

We present a novel iterative algorithm that approximates all roots of a univariate polynomial. The iteration uses floating-point eigenvalue computation of a generalized companion matrix. With some assumptions, we show that the algorithm approximates the roots to floating-point accuracy within about $\log_{\rho/\epsilon} \chi(P)$ iterations, where ϵ is the relative error of floating-point arithmetic, ρ is the relative separation of the roots, and $\chi(P)$ is the condition number of the polynomial. Each iteration requires an $n \times n$ floating-point eigenvalue computation, n the polynomial degree, and evaluation of the polynomial to floating-point accuracy at n points. On some hard examples of ill-conditioned polynomials, e.g. high-degree Wilkinson polynomials, a careful implementation of the algorithm is an order of magnitude faster than the best alternative.

1 Introduction

The algorithmic problem of approximating the roots of a univariate polynomial, presented by its coefficients, is classic in numerical analysis and symbolic algebra [17, 18]. Existing algorithms can be crudely and unexhaustively classified either as iterative algorithms or as theoretically complete algorithms. The theoretically complete algorithms include Weyl's algorithm and the Schönhage splitting-circle algorithm. The descendants of the splitting-circle algorithm (see [18] for references) have astonishingly fast asymptotic running times, at least in the real RAM model. The iterative algorithms include Newton's method, the Durand-Kerner method, and Aberth's method (see [3, 18] for references). The iterative algorithms usually have local convergence proofs, but are not known to converge globally and hence have no running time analysis.

In practice the iterative algorithms can perform very well, despite their lack of theoretical justification. Recently Bini and Fiorentino [3, 4] have developed `mpsolve`, a sophisticated and robust implementation of Aberth's method. They performed an extensive comparison of their implementation with the solvers in Maple™ and Mathematica™ and with Gourdon's implementation [12] of the Schönhage splitting-circle algorithm. Their implementation is much the fastest, beating the other implementations by factors ranging from roughly 10 to 5000, depending upon instance.

We present a novel iterative root-finding algorithm `eigensolve` and report on its performance. The algorithm, presented in section 2, bases its iteration on floating-point eigenvalue computation, and can approximate roots to (at least) floating-point accuracy. At a high level, the algorithm is very simple (see figure 1). Let P be a polynomial and S the current estimate of the roots of P, represented as floating-point numbers. The iteration is to compute the eigenvalues, in hardware floating-point, of a generalized companion matrix $C(P, S)$; these computed eigenvalues then form the next estimate of the roots of P. The matrix $C(P, S)$ is an eigenvalue formulation of Lagrange interpolation. (This matrix is familiar; a standard way to find inclusion regions that are guaranteed to contain the roots of P is to apply Gershgorin's theorem

to this matrix [3, 14].) Computation of $C(P, S)$ needs the value of P at the points of S to floating-point accuracy, requiring extended-precision arithmetic if evaluation of P is ill-conditioned.

It is not obvious that this iteration converges—indeed it might seem highly implausible, since the heart of the iteration is a floating-point eigenvalue computation on a matrix whose (eigenvalue) condition number may be vastly larger than $1/\epsilon$, ϵ the relative accuracy of hardware floating-point arithmetic. Figure 3 depicts the behavior of `eigensolve` on the Wilkinson polynomial of degree 60.

Why does the iteration converge? For a vector $S = (s_1, \ldots, s_n)$, define $Q_S(x) = \prod(x - s_i)$, and let S_0, S_1, ... be successive root estimate vectors. We observed experimentally that for each root λ of P, $|Q_{S_{i+1}}(\lambda)| \lesssim \epsilon |Q_{S_i}(\lambda)|$. See figure 4. Hence for some t, $|Q_{S_t}(\lambda)| \leq \epsilon |\lambda P'(\lambda)|$, for all roots λ of P. This is sufficient to guarantee that the root estimates in S_t are correct roughly to floating-point accuracy (lemma 3.7 below). In some sense, the successive companion matrices $C(P, S_i)$, $i = 1, 2, \ldots$, are better and better conditioned with respect to floating-point eigenvalue computation.

Section 3.2 below presents a partial correctness proof of the algorithm, requiring some assumptions. The proof applies only if P is simple, i.e. has no repeated roots. The main technical lemma, lemma 3.4, is an error analysis of floating-point eigenvalue computation on the companion matrix $C(P, S)$. It demonstrates that for each root λ of P, $|Q_{S_{i+1}}(\lambda)| \lesssim \epsilon \; p(n, \tau, 1/\rho) \, |Q_{S_i}(\lambda)|$. Here p is a low-degree polynomial, n is the degree of P, ρ is the minimum relative separation of the roots of P, and τ is discussed below. The proof of this lemma requires the technical assumption that eigenvalue computation is componentwise backwards stable—see section 2.3. The lemma implies that the algorithm converges to floating-point accuracy within roughly $\log_{\rho/\epsilon} \chi(P)$ iterations (theorem 3.5), where $\chi(P)$ is the condition number of the polynomial.

The error analysis in lemma 3.4 involves certain expressions $\tau_j(S)$ and $\sigma_j(S)$ in the roots of P and the current estimate S to the roots (section 3.1). These expressions are similar to symmetric functions and, roughly, quantify the clustering of S with respect to the clustering of the roots of P. We do not know how to give a useful *a priori* upper bound on the magnitudes of these expressions; hence we let τ be the upper bound, and state the analysis in terms of it. Empirically (section 4.8) these values are small, usually less than one. We conjecture that these values should always be small (section 3.1), but a rigorous proof is problematic.

Though the `eigensolve` algorithm is basically quite simple, and could be implemented quickly in a language like Maple, good performance requires careful attention to implementation details. Section 4 discusses the implementation of `eigensolve`, including the many forms of arithmetic, the floating-point eigenvalue solver, convergence testing, and deflation. Section 4 also compares the performance of `eigensolve` with the Bini-Fiorentino implementation `mpsolve`, using many of the test polynomials provided on the `mpsolve` web site. The `eigensolve` implementation is usually faster than `mpsolve`, with improvements of a factor of 20 for hard polynomials of degree 320, and in some cases improvements of up to a factor of 100 (see figure 5).

We believe that the algorithm and its analysis can be further refined. In particular, the analysis does not handle repeated or closely clustered roots—always the bane of iterative algorithms. The implementation has some promising but as-yet incomplete techniques to handle such roots. Furthermore, the implementation only reports roots to floating-point precision, though in principle arbitrary precision is possible. Nonetheless, the algorithm seems of sufficient interest to warrant exposure to the research community.

2 Background

Let P be a polynomial of degree n. If λ is a simple root of polynomial P, then the *condition number of P at λ* is

$$\chi(P,\lambda) = \frac{||P||\max(1,|\lambda|^{n-1})}{|P'(\lambda)|}.$$

Using standard arguments [6] it is possible to show that $\min(|\Delta\lambda|, |\Delta\lambda/\lambda|) \leq \chi(P,\lambda)\ ||\Delta P||/||P||$, where $\Delta\lambda$ is the perturbation in λ resulting from a sufficiently small perturbation ΔP to P. If P has all simple roots, then $\chi(P)$, the *condition number of P*, is the maximum of $\chi(P,\lambda)$, λ varying over the roots of P.

If P has only simple roots, then the *relative root separation* of P is

$$\mathrm{relsep}(P) = \min_{\lambda,\lambda'} \frac{|\lambda-\lambda'|}{|\lambda|+|\lambda'|}$$

where the minimum is over all distinct pairs of roots. Clearly $\mathrm{relsep}(P) \leq 1$. If λ is a simple root of P and $\alpha>0$, s is *α-close* to λ if $|s-\lambda| \leq \alpha \min_{\lambda'\neq\lambda}|\lambda-\lambda'|$, where λ' varies over all roots of P distinct from λ. The following two propositions will be needed in section 3.2.

Proposition 2.1 *If s is $1/n$-close to root λ of P, then $|P(s)| \leq e\,|s-\lambda|\ |P'(\lambda)|$.*

Proof: Suppose $P=\prod(x-\lambda_i)$ and s is α-close to λ_1. Then for $i \neq 1$, $|s-\lambda_i| = |s-\lambda_1+\lambda_1-\lambda_i| \leq (1+\alpha)\,|\lambda_1-\lambda_i|$ and

$$|P(s)| \ = \ |(s-\lambda_1)| \prod_{i\neq 1} |s-\lambda_i| \ \leq \ |s-\lambda_1|\ (1+\alpha)^{n-1}|P'(\lambda_1)|.$$

The lemma follows for $\alpha = 1/n$ using $(1+1/n)^{n-1} \leq e$. □

Proposition 2.2 *If s is not $1/n$-close to a root of λ of P, then $|s/(s-\lambda)| < (n+1)/\,\mathrm{relsep}(P)$.*

Proof: We show the contrapositive. Suppose $|s/(s-\lambda)| \geq (n+1)/\,\mathrm{relsep}(P)$. By inverting, $|(s-\lambda)/s| \leq \mathrm{relsep}(P)/(n+1)$. Since $\mathrm{relsep}(P) \leq 1$, $|(s-\lambda)/s| < 1/(n+1)$, which easily implies $|\lambda| > n/(n+1)\,|s|$. Hence for any root $\lambda' \neq \lambda$,

$$\left|\frac{s-\lambda}{\lambda}\right| \ \leq \ \frac{n+1}{n}\left|\frac{s-\lambda}{s}\right| \ \leq \ \frac{\mathrm{relsep}(P)}{n} \ \leq \ \frac{1}{n}\frac{|\lambda-\lambda'|}{|\lambda|+|\lambda'|} \ \leq \ \frac{1}{n}\left|\frac{\lambda-\lambda'}{\lambda}\right|.$$

Hence $|s-\lambda| \leq (1/n)\,|\lambda-\lambda'|$. □

For $S=(s_1,\ldots,s_k)$ a vector of complex numbers, define

$$Q_S(x) = \prod_{i=1}^{k}(x-s_i).$$

Notice that $Q'(s_j) = \prod_{i\neq j}(s_j-s_i)$.

For M a matrix, $\lambda(M)$ denotes the vector of eigenvalues of M, $\det M$ the determinant of M, and $\mathrm{char}\,M$ the characteristic polynomial of M. For monic $P = x^n + p_{n-1}x^{n-1} + \ldots + p_1x + p_0$, the *companion matrix* is

$$C(P) = \begin{bmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ & \vdots & & \ddots & \\ -p_0 & -p_1 & -p_2 & \ldots & -p_{n-1} \end{bmatrix}.$$

Standard results are that $\lambda(C(P))$ are the roots of P and that $\lambda(M)$ are the roots of $\mathrm{char}\,M$.

2.1 Some matrix facts

Throughout this section, $D = \mathrm{diag}(d_1, \ldots, d_n)$ is a diagonal matrix and $R = (r_{ij})$ is a matrix of rank one.

Lemma 2.3 $\det D+R = \prod_{i=1}^{n} d_i + \sum_{i=1}^{n} r_{ii} \prod_{j\neq i} d_j.$

Proof: The proof is by induction on n; the case $n = 1$ is immediate. Write $M[i{:}j]$ for the matrix consisting of columns i through j of matrix M, and $M[i]$ for $M[i : i]$. An easy determinant identity is $\det M = \det M_0 + \det M_1$, where M_0 and M_1 are identical to M except in the kth column, and $M_0[k]+M_1[k] = M[k]$. Using this identity,

$$\det D+R = \det \Big[D[1] \;\; (D+R)\,[2:n] \Big] + \det \Big[R[1] \;\; (D+R)\,[2:n] \Big].$$

Using the induction hypothesis,

$$\det \Big[D[1] \;\; (D+R)\,[2:n] \Big] = d_1 \Big(\prod_{i=2}^{n} d_i + \sum_{i=2}^{n} r_{ii} \prod_{j=2,\ j\neq i}^{n} d_j \Big) = \prod_{i=1}^{n} d_i + \sum_{i=2}^{n} r_{ii} \prod_{j=1,\ j\neq i}^{n} d_j.$$

Using the identity on the last column backwards to the second column,

$$\begin{aligned} \det \Big[R[1] \;\; (D+R)\,[2:n] \Big] &= \det \Big[R[1] \;\; D[2:n] \Big] + \sum_{j=2}^{n-1} \det \Big[R[1] \;\; (D+R)\,[2:j] \;\; R[j+1:n] \Big] \\ &= r_{11} \prod_{j=2}^{n} d_j \end{aligned}$$

since any matrix with at least two columns of the rank-one matrix R has zero determinant. □

Corollary 2.4 $(-1)^n \,\mathrm{char}\, D+R = \prod_{i=1}^{n} (x-d_i) + \sum_{i=1}^{n} r_{ii} \prod_{j\neq i} (x-d_i).$

Let $\det_{ij} M$ be the determinant of the matrix obtained from M by deleting the ith row and the jth column, and similarly for $\mathrm{char}_{ij} M$. The proof of the following lemma is similar to the proof of lemma 2.3.

Lemma 2.5

$$\det{}_{ij}\, D+R = \begin{cases} r_{ji} \displaystyle\prod_{k\neq i,j} d_k & i \neq j \\[2ex] \displaystyle\prod_{k\neq i} d_k + \sum_{l\neq i} r_{ll} \prod_{k\neq i,l} d_k & i = j. \end{cases}$$

Corollary 2.6

$$(-1)^{n-1} \mathrm{char}_{ij}\, D+R = \begin{cases} r_{ji} \displaystyle\prod_{k\neq i,j} (x-d_k) & i \neq j \\[2ex] \displaystyle\prod_{k\neq i} (x-d_k) + \sum_{l\neq i} r_{ll} \prod_{k\neq i,l} (x-d_k) & i = j. \end{cases}$$

2.2 Generalized companion matrices

Let $S = (s_1, \ldots, s_n)$ with s_i distinct complex numbers, and let P be a monic polynomial of degree n. The *Lagrange coefficients* $l_1, \ldots, l_n$ for P and S are defined by

$$l_i = \left| \frac{P(s_i)}{Q'_S(s_i)} \right|.$$

The *generalized companion matrix* $C(P, S)$ is the $n \times n$ matrix

$$C(P,S) \quad = \quad \begin{bmatrix} s_1 & 0 & \ldots & 0 \\ 0 & s_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \vdots & s_n \end{bmatrix} - \begin{bmatrix} l_1 & l_2 & \ldots & l_n \\ l_1 & l_2 & \ldots & l_n \\ \vdots & \vdots & & \vdots \\ l_1 & l_2 & \ldots & l_n \end{bmatrix}$$

By Corollary 2.4,

$$(-1)^n \operatorname{char} C(P,S) = \prod (x - s_i) \; + \; \sum_{i=1}^{n} l_i \prod_{j \neq i} (x - s_j).$$

The second term on the righthand side of this equation is the Lagrange interpolating polynomial [21] of degree $n-1$ that has value $P(s_i)$ at s_i. Since this polynomial is unique, and P is monic of degree of n, we have a result apparently first proved by Smith [14].

Corollary 2.7 *If P is monic, then $(-1)^n \operatorname{char}(C(P,S)) = P$; hence the eigenvalues of $C(P,S)$ are the roots of P.*

For complex s and real $r>0$, let $D(s,r) \subset \mathbb{C}$ be the closed disc of radius r centered at s. The following lemma follows easily from Gershgorin's theorem [11] and corollary 2.7 (see also [3, 9, 14]).

Lemma 2.8 *Let P be a monic polynomial of degree n, $S = (s_1, \ldots, s_n)$ be distinct complex numbers and $l_1, \ldots, l_n$ be the Lagrange coefficients. Then the union of discs $\bigcup_{i=1}^{n} D(s_i, nl_i)$ contains all the roots of P. Furthermore, each connected component made up of k discs contains k roots. In particular, if the discs are pairwise disjoint, each disc contains exactly one root.*

2.3 Floating-point eigenvalue computation

The computation of eigenvalues has been a central topic of numerical analysis for several decades. The QR algorithm[7, 11] is the standard for dense matrices. The conventional analysis is that the QR algorithm is backwards stable: for M an $n \times n$ matrix with floating-point entries, a call to the implementation `eig` of the QR algorithm

$$\Lambda := \texttt{eig}(M)$$

computes a vector of eigenvalues $\Lambda = (\lambda_1, \ldots, \lambda_n)$ that are the true eigenvalues of a perturbed matrix $M + \Delta M$:

$$\begin{aligned} \Lambda &= \lambda(M + \Delta M) \\ ||\Delta M|| &\leq t(n)\, \epsilon\, ||M||, \end{aligned}$$

```
// P is a monic polynomial of degree n
// S = (s_1, ..., s_n) is a vector of length n of floating-point complex numbers
// C(P) and C(P, S) are companion matrices
// fl is conversion to floating-point
// eig is floating-point eigenvalue calculation (section 2.3)
procedure eigensolve (P)
    S := eig(fl(C(P)))
    while ¬converged(S)
            S := eig(fl(C(P, S)))
    return S
```

Figure 1: The iterated eigenvalue algorithm. Convergence is discussed in the text.

where ϵ is the relative error of floating-point arithmetic and the *growth factor* $t(n)$ is bounded by a low-degree polynomial in n. Apparently there is no known rigorous upper bound on $t(n)$; it is usually cited as a slowly-growing function of n, say $t(n) \leq 10n$ [1].

In section 3.2 we assume that `eig` is *componentwise backwards stable*, that is, the matrix ΔM satisfies $|\Delta M_{ij}| \leq p(n)\,\epsilon\,|M_{ij}|$, for some polynomial $p(n)$. This assumption is not true in general, and we cannot give any rigorous justification for it in this context. However, the assumption significantly simplifies the error analysis in section 3.2, and the error analysis does appear to reflect the true behavior of the `eigensolve` algorithm. It is possible that the assumption could be eliminated with a more complex analysis exploiting the special structure of the matrices used in `eigensolve`.

3 The algorithm and its analysis

For this section we assume that P is monic of degree n, with distinct nonzero roots $\lambda_1, \ldots, \lambda_n$. The algorithm for root finding is given in figure 1. Implementation details of the algorithm, including a discussion of convergence, are given in section 4.

3.1 The expressions σ and τ

Suppose $S = \{s_1, \ldots, s_n\}$ are distinct complex numbers. With λ_k the root of P closest to s_j, define

$$\begin{aligned} \tau_j(S) &= \frac{\prod_{i \neq k} s_j - \lambda_i}{\prod_{i \neq j} s_j - s_i} \\ \sigma_j(S) &= \prod_{i \neq j} \frac{\lambda_k - s_i}{s_j - s_i}. \end{aligned} \tag{1}$$

Conjecture 3.1 *Let P be arbitrary and let G be a random polynomial with $||G|| \ll ||P||$. Let S be the roots of $P+G$. Then with high probability, for $j = 1, \ldots, n$,*

$$|\tau_j(S)| = O(1),$$

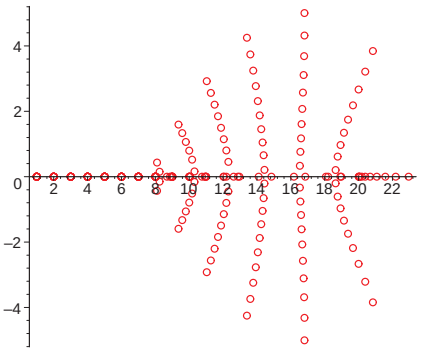

Figure 2: Roots of $W_{20} + \epsilon G$, G random of degree 19 with unit norm, $\epsilon = 10^{-10} \ldots 10^{-5}$.

and as long as s_j is $1/n$-close to some root of P,

$$|\sigma_j(S)| = O(1).$$

The conjecture is not intended to be completely precise: "$O(1)$" is meant to be read as "small," in fact around one, though depending perhaps upon the degree of P but not its coefficients. The exact distribution on G and the meaning of "$\ll$" would be determined only by a proof of the conjecture. The conjecture is most interesting when P is ill-conditioned, since in this case the roots of $P+G$ can differ significantly from the roots of P.

The quantities τ_j can be interpreted as follows. If P is ill-conditioned, then P has clustered roots. However, S, the roots of $P+G$, are relatively spread apart and more-or-less uniformly spaced. Figure 2 depicts the case where P is the Wilkinson polynomial of degree 20. Then $|\tau_j|$ can be interpreted as the ratio between two products: the product of distances from s_j to all the roots of P except λ_k, and the product of distances from s_j to all s_i, $i \neq j$. Since S is "spread-out" and "uniformly spaced" with respect to the roots of P, it is plausible that the ratio should be small.

The quantities σ_j have a similar, slightly simpler interpretation. Suppose G is a perturbation so that a root s_j of $P + G$ happens to be close to root λ_k of P. Random perturbations tend to "spread out" roots, so any other root s_i will move away from λ_k and s_j. Hence $\lambda_k - s_i \lesssim s_j - s_i$, and σ_j is small.

The following lemmas relate the conjecture to the analysis below.

Proposition 3.2 *For all $i, j = 1, \ldots, n$, $|l_j/(\lambda_i - s_j)| \leq |\tau_j|$.*

Proof: Let λ_k be the root of P closest to s_j. Then

$$\left|\frac{l_j}{\lambda_i - s_j}\right| \leq \left|\frac{l_j}{\lambda_k - s_j}\right| = \left|\frac{P(s_j)}{Q'(s_j)\,(\lambda_k - s_j)}\right| = |\tau_j|.$$

□

Lemma 3.3 *Let $M = C(P, S)$ and λ_k be the root of P closest to s_i. Then* $(\mathrm{char}_{ii}\, M)(\lambda_k) = \dfrac{P(s_i)}{\lambda_k - s_i}\sigma_i$.

Proof: Using corollaries 2.4 and 2.6 and the definition of M,

$$\mathrm{char}\, M = (x - s_i)\ \mathrm{char}_{ii}\, M + l_i \prod_{j \neq i} x - s_j.$$

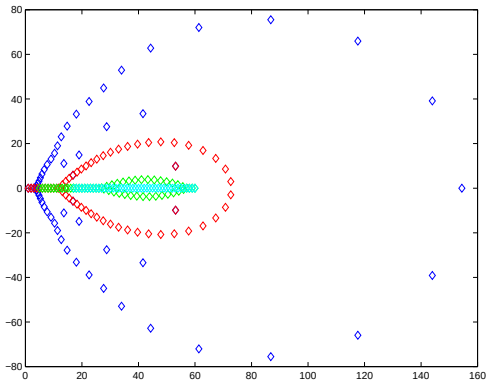

Figure 3: Iterates of the algorithm on the Wilkinson polynomial of degree 60.

Substituting $x = \lambda_k$, we have $(\operatorname{char} M)(\lambda_k) = P(\lambda_k) = 0$ and

$$\begin{aligned}(\operatorname{char}_{ii} M)(\lambda_k) &= \frac{l_i}{\lambda_k - s_i} \prod_{j \neq i} \lambda_k - s_j \\ &= \frac{P(s_i)}{\lambda_k - s_i} \prod_{j \neq i} \frac{\lambda_k - s_j}{s_i - s_j}.\end{aligned}$$

□

The analysis below requires upper bounds on $|\sigma_j(S)|$ and $|\tau_j(S)|$, where S is an iterate of `eigensolve`. Since `eig` is backwards stable, each vector S is the vector of eigenvalues of a slightly perturbed matrix $M + \Delta M$, where $\operatorname{char} M = P$. The hope is that the perturbation due to floating-point rounding behaves like a random perturbation. The statement of conjecture 3.1 was chosen to be simple, but it is not by itself sufficient. First, conjecture 3.1 is expressed in terms of perturbations of the coefficients of P, not the entries in M. Second, floating-point perturbations are far from random. The perturbation matrix ΔM results from the rounding that occurred during many floating-point arithmetic operations. Even if each such individual rounding were random, the entries in ΔM would not be independent, but would have complex correlations due to the structure of the eigenvalue algorithm. Hence it is problematic even to state a conjecture strong enough for the analysis of `eigensolve`.

3.2 Analysis

Let S_0 be the values computed by the initial call `eig(fl`$(C(P))$`)`, and for $t = 1, 2, \ldots$, let S_t be the values computed at the tth call to `eig` within the `while` loop. With

$$S_t = \{s_{t1}, \ldots, s_{tn}\}$$

define

$$Q_t(x) = Q_{S_t}(x) = \prod_{i=1}^{n} (x - s_{ti}).$$

Since it is unclear how to give a useful *a priori* upper bound on $|\sigma_j(S_t)|$ and $|\tau_j(S_t)|$ the analysis is given with an upper bound appearing explicitly. Let C be the pairs (j, t) so that s_{tj} is $1/n$-close to a root of P, and define

$$\tau = \max\left(\max_{j,t} |\tau_j(S_t)|, \max_{(j,t) \in C} |\sigma_j(S_t)| \right). \tag{2}$$

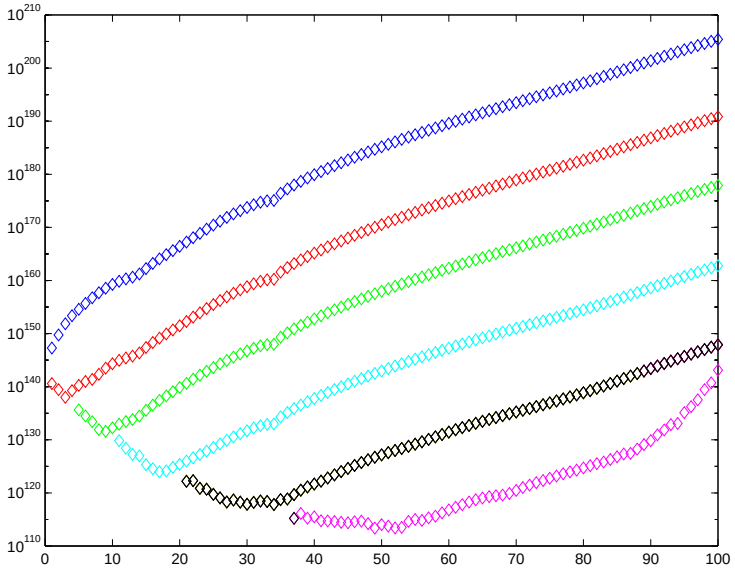

Figure 4: The x-axis are the roots 1..100 of the Wilkinson polynomial of degree 100; dots are the values of $Q_i(\lambda)$, i varying over the iteration. The lower envelope of the dots closely matches $\epsilon|\lambda W'_{100}(\lambda)|$.

Section 4.8 discusses observed behavior of τ.

The following lemma is the main technical lemma describing the iteration in `eigensolve`.

Lemma 3.4 *Assume τ is defined by equation (2) and that* `eig` *is componentwise backwards stable. For each root λ of P,*

$$|Q_{t+1}(\lambda)| \;\leq\; \epsilon\, p(n,\tau)\, \max\Big(\, |\lambda P'(\lambda)|,\; |Q_t(\lambda)|/\operatorname{relsep}(P)\,\Big).$$

where $p(n,\tau)$ is a polynomial in n and τ.

Figure 4 depicts lemma 3.4 for the Wilkinson polynomial of degree 100. The x-axis are the roots, i.e. the integers from 1 to 100. Each dot is a value $Q_i(\lambda)$, λ a root, i the index of an iteration. Notice that the dots along a vertical line are separated by about $\epsilon \approx 10^{-16}$.

The lower bound in lemma 3.4 can be motivated very simply. Let $R = (r_1, \ldots, r_n)$ where r_i is the best floating-point approximation to λ_i. Then for $k = 1, \ldots, n$, $|r_k - \lambda_k| \leq \epsilon|\lambda_k|$ and

$$|Q_R(\lambda_k)| = |(\lambda_k - r_k)\prod_{j\neq k}(\lambda_k - r_j)| \approx |\epsilon\lambda_k \prod_{j\neq k}(\lambda_k - \lambda_j)| = |\epsilon\lambda_k P'(\lambda_k)|.$$

The lower envelope of the dots in figure 4 closely matches $\epsilon\lambda P'(\lambda)$.

The following theorem is a consequence of lemma 3.4. We prove it using the easy lemmas 3.6 and 3.7. Then we give the proof of lemma 3.4.

Theorem 3.5 *Assume τ is defined by equation (2) and that* `eig` *is componentwise backwards stable. There is a polynomial $q(n,\tau)$ of n and τ, so that with*

$$\begin{aligned} f &= \lceil \log_\beta \chi(P) \rceil \\ \beta &= \frac{\operatorname{relsep}(P)}{\epsilon\; q(n,\tau)} \end{aligned}$$

we have that S_f approximates the roots of P almost to floating-point precision. In particular, there is a permutation σ on $\{1,\ldots,n\}$ so that for $i = 1,\ldots,n$,

$$|\lambda_i - s_{f\sigma(i)}| \leq n^{O(1)}\,\epsilon \max(1,|\lambda_i|).$$

Lemma 3.6 *For each root λ of P, $|Q_0(\lambda)| \leq n^{O(1)}\,\epsilon||P||\,\max(1,|\lambda|^n)$.*

Proof: By the assumptions on the behavior of `eig`, $S_0 = \lambda(\tilde{M})$ where $\tilde{M} = C(P)+\Delta M$, $\Delta M \leq n^{O(1)}\,\epsilon||C(P)||$. By a result of Van Dooren and Dewilde [20] (see also [8]), $\lambda(\tilde{M}) = \lambda(C(\tilde{P}))$ where $\tilde{P} = P+\Delta P$, $||\Delta P|| \leq n^{O(1)}\,\epsilon||P||$. But $\tilde{P} = Q_0$, since they are both monic polynomials with the same roots. Hence $|Q_0(\lambda)| = |P(\lambda)+\Delta P(\lambda)| = |\Delta(P)(\lambda)|$. The lemma follows using $|\Delta P(\lambda)| \leq n||\Delta P||\max(1,|\lambda|^n)$. □

Lemma 3.7 *Suppose at the tth iteration, each root λ of P satisfies*

$$|Q_t(\lambda)| \leq n^{O(1)}\,\epsilon \max(1,|\lambda|)|P'(\lambda)|.$$

Then S_t approximates the roots of P almost to floating-point precision, as in theorem 3.5.

Proof: Apply lemma 2.8 with the role of P taken by Q_t and Q by P. Then S_t is contained in the union $\cup D(\lambda, n^{O(1)}\,\epsilon|\lambda|)$, λ varying over the roots of P. Furthermore each connected component of the union consisting of k discs contains exactly k elements of S_t. The lemma follows easily, since any component of k discs is contained in a single disc of radius about k times bigger than any individual disc. □

Proof of Theorem 3.5: The ratio of the bounds in lemmas 3.6 and 3.7 satisfies

$$\frac{n^{O(1)}\,\epsilon||P||\,\max(1,|\lambda|^n)}{n^{O(1)}\,\epsilon\max(1,|\lambda|)|P'(\lambda)|} \leq n^{O(1)}\chi(P).$$

The bound on f follows from lemma 3.4. □

Proof of Lemma 3.4: Consider a root λ_k of P. For simplicity drop the double subscripts and write $S_t = (s_1,\ldots,s_n)$. Let $M = C(P,S_t)$. Recall that $M_{ij} = -l_j$ if $i \neq j$, and $M_{jj} = s_j - l_j$, where $l_j = P(s_j)/Q_t'(s_j)$ is the Lagrange coefficient.

By the assumptions on `eig`, there is a perturbation matrix Δ so that

$$\begin{aligned} S_{t+1} &= \lambda(M+\Delta) \\ Q_{t+1}(x) &= \mathrm{char}(M+\Delta) \\ \Delta_{ij} &= \delta_{ij}\,M_{ij}, \qquad |\delta_{ij}| \leq n^{O(1)}\,\epsilon. \end{aligned} \tag{3}$$

By expanding the determinant and letting $O(\Delta^2)$ denote the sum of all terms with at least two factors from Δ, we get

$$Q_{t+1}(x) \;=\; \mathrm{char}\,M \;+\; \sum_{i,j}(-1)^{i+j}\Delta_{ij}\,\mathrm{char}_{ij}\,M \;+\; O(\Delta^2);$$

of course $\mathrm{char}\,M = P$. We henceforth ignore the $O(\Delta^2)$ terms; this can be justified rigorously by a somewhat tedious argument similar to the argument below. Substituting λ_k for x, we get

$$|Q_{t+1}(\lambda_k)| \;\leq\; \sum_{i,j}|\Delta_{ij}\;(\mathrm{char}_{ij}\,M)(\lambda_k)|. \tag{4}$$

If $i \neq j$ then by corollary 2.6 and using $M_{ij} = -l_j$,

$$\begin{aligned}
\operatorname{char}_{ij} M &= M_{ji} \prod_{k \neq i,j} (x - s_k) \\
\Delta_{ij} \operatorname{char}_{ij} M &= \delta_{ij} M_{ij} M_{ji} \prod_{k \neq i,j} (x - s_k) \\
&= \delta_{ij} \frac{l_j}{x - s_j} \frac{l_i}{x - s_i} Q_t(x) \\
\Delta_{ij} \ (\operatorname{char}_{ij} M)(\lambda_k) &= \delta_{ij} \frac{l_j}{\lambda_k - s_j} \frac{l_i}{\lambda_k - s_i} Q_t(\lambda_k) \\
|\Delta_{ij} \ (\operatorname{char}_{ij} M)(\lambda_k)| &\leq |\delta_{ij}| \, \tau^2 |Q_t(\lambda_k)|, \qquad (5)
\end{aligned}$$

where the last line uses proposition 3.2 and equation (2).

Now suppose $i = j$. We have $M_{ii} = s_i - l_i$. There are two cases depending on whether or not s_i is $1/n$-close to λ_k. Suppose s_i is not $1/n$-close to λ_k; we have

$$\begin{aligned}
\operatorname{char}_{ii} M &= \prod_{j \neq i} (x - s_j) \ + \sum_{p \neq i} l_p \prod_{j \neq i,p} (x - s_j) \\
&= \Big(1 + \sum_{p \neq i} \frac{l_p}{x - s_p}\Big) \prod_{j \neq i} (x - s_j).
\end{aligned}$$

Substituting $x = \lambda_k$, using $\Delta_{ii} = \delta_{ii}(s_i - l_i)$, and taking absolute values yields

$$\begin{aligned}
|\Delta_{ii} \, (\operatorname{char}_{ii} M)(\lambda_k)| &\leq |\delta_{ii}| \left(\left| \frac{s_i}{\lambda_k - s_i} \right| + \left| \frac{l_i}{\lambda_k - s_k} \right| \right) \left(1 + \sum_{p \neq i} \left| \frac{l_p}{\lambda_k - s_p} \right| \right) \, |Q_t(\lambda_k)| \\
&\leq \left(\frac{n+1}{\operatorname{relsep}(P)} + \tau \right) \, (1 + n\tau) \, |Q_t(\lambda_k)|, \qquad (6)
\end{aligned}$$

where the last line uses propositions 2.2 and 3.2 and equation (2).

Finally, suppose $i = j$ and s_i is $1/n$-close to λ_k. Again $M_{ii} = s_i - l_i$, so $|\Delta_{ii}| \leq |\delta_{ii} \, s_i| + |\delta_{ii} \, l_i|$. The term involving $|\delta_{ii} \, l_i|$ can be agued as above. Since s_i is $1/n$-close to λ_k, λ_k is the root of P closest to s_i. Using lemma 3.3 and then proposition 2.1,

$$|\delta_{ii} \, s_i \, (\operatorname{char}_{ii} M)(\lambda_k)| \ \leq \ \left| \delta_{ii} \, s_i \, \frac{P(s_i)}{\lambda_k - s_i} \, \sigma_i \right| \ \leq \ \kappa \, |\delta_{ii} \, \lambda_k \, P'(\lambda_k)|. \qquad (7)$$

for a small constant $\kappa > 0$. Lemma 3.4 now follows from (3), (4), (5), (6), and (7). □

4 Implementation and performance

The algorithm has been implemented in C++ using EISPACK [15] for floating-point eigenvalue computation and GMP [13] for extended-precision arithmetic. The source code, exclusive of libraries, is about 2000 lines of C++. Some of the details of the implementation are discussed below.

4.1 Arithmetic types

The implementation uses three arithmetic types to represent real numbers: the native C++ `double` type, a class `dpe`, and a class `LongDouble`. Each of these types has a complex analog as well. Assuming IEEE arithmetic, a `double` value is at most 2^{1024} and the relative error of any arithmetic operation on `doubles` is at most $\epsilon{=}2^{-53}{\approx}10^{-16}$. The `dpe` class [2, 3] contains a `double` d, with $d{=}0$ or $.5 \leq |d| \leq 1$, and an `int` i, representing the number $d \cdot 2^i$. It allows representation of real numbers with the precision of a `double` but with very large dynamic range. C++ operator overloading provides convenient syntax for the usual arithmetic operations. The `LongDouble` class provides extended-precision floating-point arithmetic with error bound accumulation, with the maximum bit-length of any result specifiable on a per-operation basis. This class uses GMP integers (type `mpz_t`) for arithmetic operations. Only assignment and update operators (`+=`,`*=`, etc.) are provided; this simplifies the implementation of the class and minimizes GMP memory allocation, though slightly complicating use of the class.

4.2 The initial companion matrix

Let $P(x) = x^n + p_{n-1}x^{n-1} + \ldots + p_0$. Construction of the initial companion matrix for P appears to be simple, just negating the coefficients. However, any eigenvalue solver from a standard numeric library requires a matrix with `double` (or `complex`) entries, and the coefficients may have magnitude too large for a `double`. The companion matrix can be constructed instead with the coefficients of $Q(y) = P(ay)/a^n = y^n + (p_{n-1}/a)\,y^{n-1} + \ldots + p_0/a^n$, where the scaling factor $a{>}0$ is chosen so that all coefficients have magnitude smaller than the maximum allowed `double`. In fact, a useful heuristic is to choose a somewhat smaller upper bound, to minimize the chance of overflow in the eigenvalue solver.

To reduce the number of iterations of the algorithm, it turns out to be useful to shift the polynomial so that the center of mass of its roots is the origin. The center of mass of the roots $c = (\lambda_1{+}\ldots{+}\lambda_n)/n$ is easily computed by $c = -p_{n-1}/n$; the shift is accomplished by computing the coefficients of $R(x) = P(x{+}c)$. The shifted coefficients can then be scaled as above. Of course, the inverse shift and scale needs to be applied to the values computed by the eigensolver.

4.3 The generalized companion matrix

The construction of the generalized companion matrix $C(P, S)$ demonstrates the rather intricate roles of the various arithmetic types. At the start of an iteration, the current estimate $S = (s_1, \ldots, s_n)$ of the roots is the result of a previous eigenvalue computation; hence each s_i is represented as a C++ `complex`, i.e. a pair of `doubles`. For the iteration, the s_i are treated as exact.

The entries of $C(P, S)$ must be represented with the `complex` type and should be computed with relative error roughly ϵ. The entries require the Lagrange coefficients $l_i = P(s_i)/\prod_{j\neq i}(s_i{-}s_j)$. The precision of `double` arithmetic is sufficient for the computation $\prod_{j\neq i}(s_i{-}s_j)$, but the magnitude of the expression may be too large. Hence the product is computed using the `dpe` type.

The computation of $P(s_i)$ may involve massive cancellation; hence the `LongDouble` class is required to get its value with relative error less than ϵ. Since the required arithmetic precision is not known *a priori*, the value $P(s_i)$ is computed using a standard doubling strategy, that is, using b, $2b$, $4b$, ... bits of precision, until the relative error in the result is at most ϵ. The initial value of b is determined by *precision caching*, i.e. by using the precision required for evaluation of $P(t_i)$, where t_i is the root estimate of the preceding iteration corresponding to s_i. The correspondence of root estimates between iterations is determined using a simple heuristic—the index of the roots in a lexicographic sort.

In principle, the entries of matrix $C(P,S)$ may have to be scaled, as in section 4.2, so that all entries have magnitude less than the maximum allowed double. In practice, for the polynomials examined so far, the ratio $P(s_i)/\prod_{j\neq i}(s_i-s_j)$ always fits in a `double`, even if the numerator and denominator do not.

4.4 Deflation

After some number of iterations, some subset of the root estimates will have converged using the criteria described in section 4.5 below. If not all root estimates have converged, it is useful to deflate the polynomial to reduce the computational cost of subsequent iterations. Suppose $s_1,\ldots,s_k$ are converged approximations of the corresponding roots $\lambda_1,\ldots,\lambda_k$ of P. Let $\tilde{S}=(s_{k+1},\ldots,s_n)$ and $\tilde{P}=P/\prod_{i=1}^{k}(x-\lambda_i)$. The computation of the $(n-k)\times(n-k)$ matrix $C(\tilde{S},\tilde{P})$ seems to require

$$\tilde{l}_i=\frac{\tilde{P}(s_i)}{\prod\limits_{j=k+1,j\neq i}^{n}(s_i-s_j)},\quad i=k+1,\ldots,n.$$

However it suffices to use the approximation $\tilde{l}_i\approx l_i$, since the ratio

$$\frac{l_i}{\tilde{l}_i}=\prod_{j=1}^{k}\frac{s_i-\lambda_j}{s_i-s_j}=\prod_{j=1}^{k}\frac{(s_i-s_j)+(s_j-\lambda_j)}{s_i-s_j}$$

is very close to 1 provided $|s_j-\lambda_j|$ is very small compared to $|s_i-s_j|$, $i>k$ and $j\leq k$. The current implementation simply ensures that $|s_j-\lambda_j|/|s_j|\leq\epsilon$, $j=1,\ldots,k$. A more rigorous implementation could use Newton iteration on demand to compute s_j with sufficient accuracy to guarantee that the ratio $l_i/\tilde{l}_i$ is within roughly ϵ of 1.

4.5 Convergence

Let $s_1,\ldots,s_n$ be the current root estimates and $l_1,\ldots,l_n$ be the Lagrange coefficients computed as described in section 4.3. By lemma 2.8, any connected component of the union $\bigcup_{i=1}^{n}D(s_i,nl_i)$ consisting of k discs contains k roots of P. A root estimate s_i is *isolated* if its disc $D(s_i,nl_i)$ is disjoint from the other discs. A simple strategy would be to declare that a root has converged if it is isolated (actually, the disc should be small enough to guarantee convergence of Newton's iteration [19]). Then the polynomial can be deflated as described above. In practice a frequent occurrence is that a disc $D(s_i,nl_i)$ has small radius and contains a root, but intersects some other larger disc. A more aggressive convergence strategy sometimes still allows deflation.

By standard results [16], for any complex t, the disc $D(t,r(t))$, $r(t)=|nP(t)/P'(t)|$, always contains at least one root of P. A set of root estimates $t_1,\ldots,t_k$, $k\leq n$, are *tentatively isolated* if the discs $D(t_1,r(t_1))$, $\ldots$, $D(t_k,r(t_k))$ are pairwise disjoint (and small enough to guarantee Newton convergence). The algorithm maintains a set of tentatively isolated root estimates; these are updated to floating-point accuracy using Newton iteration. If all root estimates are tentatively isolated, the algorithm terminates. If not, the polynomial is deflated by the tentatively isolated root estimates, and the eigenvalue iteration continues with the remaining root estimates, called *active* estimates.

How does an active root estimate become tentatively isolated? The following heuristic avoids frequent evaluation of the derivative P' (which is computationally expensive) by using the Lagrange coefficients (which are computed anyway) as a hint. An active root estimate s_i is a *candidate* if its disc $D(s_i,nl_i)$ is

relatively small, in particular $nl_i \leq \sqrt{\epsilon} \min_{j \neq i} |s_j - s_i|$, and the only intersection of disc $D(s_i, nl_i)$ with other active discs is with active discs of much larger radius. If s_i is a candidate, the radius $r(s_i)$ is computed; if this radius is not relatively small, s_i is no longer considered a candidate. Otherwise the disc $D(s_i, r(s_i))$ is compared with the discs of all other other tentatively isolated root estimates. If there is no intersection, s_i becomes tentatively isolated. If there is an intersection (which is very rare), all root estimates that are involved in the intersection are marked active. The last step is necessary (and motivates the term "tentative isolation") since for example there is no guarantee that there is only one root in the disc $D(s_i, r(s_i))$ of tentatively isolated estimate s_i.

4.6 Matrix simplification

Sometimes a Lagrange coefficient l_i for an active root s_i has small magnitude (usually because $r(s_i)$ is large), while other Lagrange coefficients are large. As a heuristic, if a Lagrange coefficient l_i satisfies $|l_i| \leq \kappa\epsilon \max_j |l_j|$, then the ith column is eliminated from the matrix (recall that the off-diagonal entries are l_i; the diagonal entry is $s_i - l_i$.). Since the eigensolve time is cubic in the matrix size, the eigensolve time can be substantially reduced if even a small fraction of the columns are eliminated. Heuristically, the column elimination can be justified as a perturbation of the matrix of relative magnitude $\kappa\epsilon$, or equivalently as increasing the relative error in arithmetic to $\kappa\epsilon$. The number of iterations increases with κ but the time per iteration decreases. We use $\kappa \approx 100$.

4.7 Performance

Figure 5 compares the running times of `eigensolve` and the Bini-Fiorentino implementation `mpsolve` on some sample polynomials. For `eigensolve`, running time is roughly balanced between floating-point eigenvalue computation and polynomial evaluation, with the other costs of the algorithm negligible. For the polynomials in figure 5, the number of iterations varied roughly between $n/25$ and $n/100$, n the degree.

The optimizations discussed above contribute significantly to the speed of `eigensolve`. For example, for the Wilkinson polynomial of degree 320, solve time increased roughly by a factor 1.9 without shifting, 1.8 without precision caching, 1.25 without matrix simplification, and 1.3 without deflation. Without any of these optimizations, solve time increased by a factor of 5.4 to about 600 seconds. Shifting is very beneficial in some cases: for the contrived "Shifted Wilkinson" polynomial in figure 5, shifting reduces the number of iterations from 22 to 2. Plausibly shifting would be a beneficial optimization for `mpsolve` as well.

4.8 The behavior of τ and σ

For many of the polynomials in figure 5, all values of τ_j and σ_j were computed at each iteration of the execution of `eigensolve`. Over all polynomials and all iterations, the per-iteration median value of τ_j and σ_j never exceeded 1. The behavior for the Wilkinson polynomial of degree 400 was typical. The algorithm required 10 iterations to converge. At the first iteration, the median values of τ_j and σ_j were about 10^{-3} and 10^{-14}, respectively. The median value of τ_j increased slowly to 1 at the penultimate iteration; the median value of σ_j increased more quickly, also ending at 1. The maximum value of τ_j was about 10^{12} at the first iteration; for subsequent iterations the maximum value was less than 3.

The large maximum at the first iteration appears to be a consequence of somewhat-mysterious behavior of the floating-point eigensolver. The initial companion matrix formulation is extremely ill-conditioned.

Polynomial	deg	mpsolve	eigen	ratio
Wilkinson	40	1.1	.2	5.5
	80	12.3	1.9	6.5
	160	144.0	12.1	11.9
	320	2138.0	107.5	19.9
	640		1448.0	
Laguerre	80	5.1	1.6	3.2
	160	59.0	11.1	5.3
	320	806.0	89.0	9.1
Chebyshev	160	39.0	12.0	3.2
	320	450.0	134.7	3.3
Chromatic a	342	1150.0	66.6	17.3
Chromatic ad	340	1848.0	81.5	22.7
Chromatic cd	171	123.1	11.5	10.7
	683	>15750.0	766.2	>20.6
Hermite	160	20.4	8.1	2.5
	320	230.0	78.8	2.9
Mandelbrot	127	11.4	3.9	2.9
	255	134.0	29.1	4.6
	511	1874.0	226.6	8.3
	1023		3390.0	
Sendra	160	291.0	23.5	12.4
Exponential	50	.3	.5	.6
	100	3.8	2.5	1.5
	200	43.0	16.7	2.6
Toeplitz 1	128	5.8	4.2	1.4
	256	60.2	31.6	1.9
Toeplitz 2	128	16.3	6.2	2.6
	256	190.4	49.25	3.8
Shifted Wilk.	100	508.0	5.3	95.8
$(300x-1)^{100}$	100	260.0	2.1	123.8

Figure 5: Comparison of `eigensolve` and `mpsolve` (version 2.1, January 2000). Times are seconds on a 250MHz SGI R10000. All but the last two polynomials are from the `mpsolve` web site. The "Shifted Wilkinson" polynomial has integer roots $10^4+1, \ldots, 10^4+100$.

In such cases, the eigensolver sometimes returns a vector of eigenvalues with identical or nearly-identical entries, which can cause the denominator of (1) to be very small.

5 Conclusion

The conditioning of many classic numerical problems, e.g. the computation of eigenvalues and the solution of linear systems, has been studied extensively in numerical analysis. For many such problems backwards-stable algorithms are known; such algorithms are the best possible if numeric input is only known approximately. In some cases, an input instance may be known exactly. If the instance is ill-conditioned, the backwards-stable algorithm does not provide an accurate solution using standard (hardware) floating-point precision. In principle the backwards-stable algorithm would suffice if arithmetic were sufficiently accurate, but in general this requires expensive software extended-precision arithmetic.

The algorithm in this paper uses an iterative conditioning technique to obtain a forward-stable algorithm. Given an instance that is known exactly, but is perhaps ill-conditioned, the problem is first solved using a standard backwards-stable floating-point algorithm. The result of the floating-point computation then is used to compute a new instance which is better conditioned but has the same solution as the original instance. This iteration is repeated until a well-conditioned instance is obtained, and the solution to it can be returned.

This iterative conditioning technique appears very promising. In addition to the algorithm in this paper, the technique has been applied to the sign-evaluation of the determinant of integer matrices [5] and to the computation of the inertia of symmetric integer matrices [10]. Roughly, the technique replaces a single computation which requires an n-fold increase in arithmetic precision with n computations, each of which requires standard arithmetic. Application of the technique is not trivial, since considerable cleverness may be required to use the result of a floating-point computation to improve the conditioning of an instance. It is worth exploring whether this technique applies to other numerical problems.

Acknowledgements. Thanks to Dario Bini and Giuseppe Fiorentino for providing a public web site with `mpsolve` and many test polynomials.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, D. Sorenson, *LAPACK User's Guide*, Second Edition, SIAM, 1995.

[2] D.H. Bailey, A portable high performance multiprecision package, RNR Technical report RNR-90-022, NAS Applied Research Branch, NASA Ames Research Center.

[3] D.A. Bini, G. Fiorentino, Design, analysis, and implementation of a multiprecision polynomial rootfinder, manuscript, 1999.

[4] D.A. Bini, G. Fiorentino, Numerical computation of polynomial roots using MPSolve, manuscript, 1999. Software and paper available for download from `ftp://ftp.dm.unipi.it/pub/mpsolve/`.

[5] K. Clarkson, Safe and effective determinant evaluation, *Proc. 33rd Annual IEEE Symp. on Found. of Computer Science*, pp. 387–395, 1992.

[6] J. Demmel, On condition numbers and the distance to the nearest ill-posed problem, *Numerische Mathematik*, **51**:251–289, 1987.

[7] J. Demmel, *Applied Numerical Linear Algebra*, SIAM, 1997.

[8] A. Edelman, H. Murakami, Polynomial roots from companion matrix eigenvalues, 1994. *Mathematics of Computation*, **64**:763–776, 1995.

[9] L. Elsner, A remark on simultaneous inclusions of the zeros of a polynomial by Gershgorin's theorem, *Numer. Math.*, **21**:425–427, 1973.

[10] S. Fortune, Exact computation of the inertia of a symmetric integer matrix , *STOC 2000*, pp. 556–564.

[11] G.H. Golub, C.F. Van Loan, *Matrix Computations*, 2nd edition, The Johns Hopkins University Press, 1990.

[12] X. Gourdon, *Combinatoire, Algorithmique et Géométrie des Polynômes*, Ph.D. thesis, L'École Polytechnique, 1996, available at `pauillac.inria.fr/algo/gourdon/`.

[13] T. Granlund, GNU MP: the GNU multiple precision arithmetic library, Edition 2.0, 1996. Code available for download from `www.swox.com/gmp/`.

[14] B.T. Smith, Error bounds for the zeroes of a polynomial based upon Gershgorin's theorem, *JACM*, **17**:661-674, 1970.

[15] B.T. Smith, J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema, C.B. Moler, Matrix Eigensystem Routines - EISPACK Guide, volume 6 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1976. Source code is available from netlib at `www.netlib.org`.

[16] P. Henrici, *Applied and computational complex analysis: Vol. 1*, John Wiley and sons, 1974.

[17] J. McNamee, A bibliography on roots of polynomials, *J. Comput. Appl. Math* **47** pp. 391–394, 1993. An updated bibliography is available at `www.elsevier.nl/homepage/sac/cam/mcnamee/index.html`.

[18] V. Pan, Solving a polynomial equation: some history and recent progress, *SIAM Review* **39**:2, pp. 187–220, 1997.

[19] P. Tilli, Convergence conditions of some methods for the simultaneous computation of polynomial zeroes, *Calcolo* **35**, pp. 3–15, 1998.

[20] P. Van Dooren, P. Dewilde, The eigenstructure of an arbitrary polynomial matrix: computational aspects, *Lin. Alg. Appl*, **50**:545–579, 1983.

[21] C.K. Yap, *Fundamental problems of algorithmic algebra*, Oxford University Press, Oxford, 2000.